SCENARIO

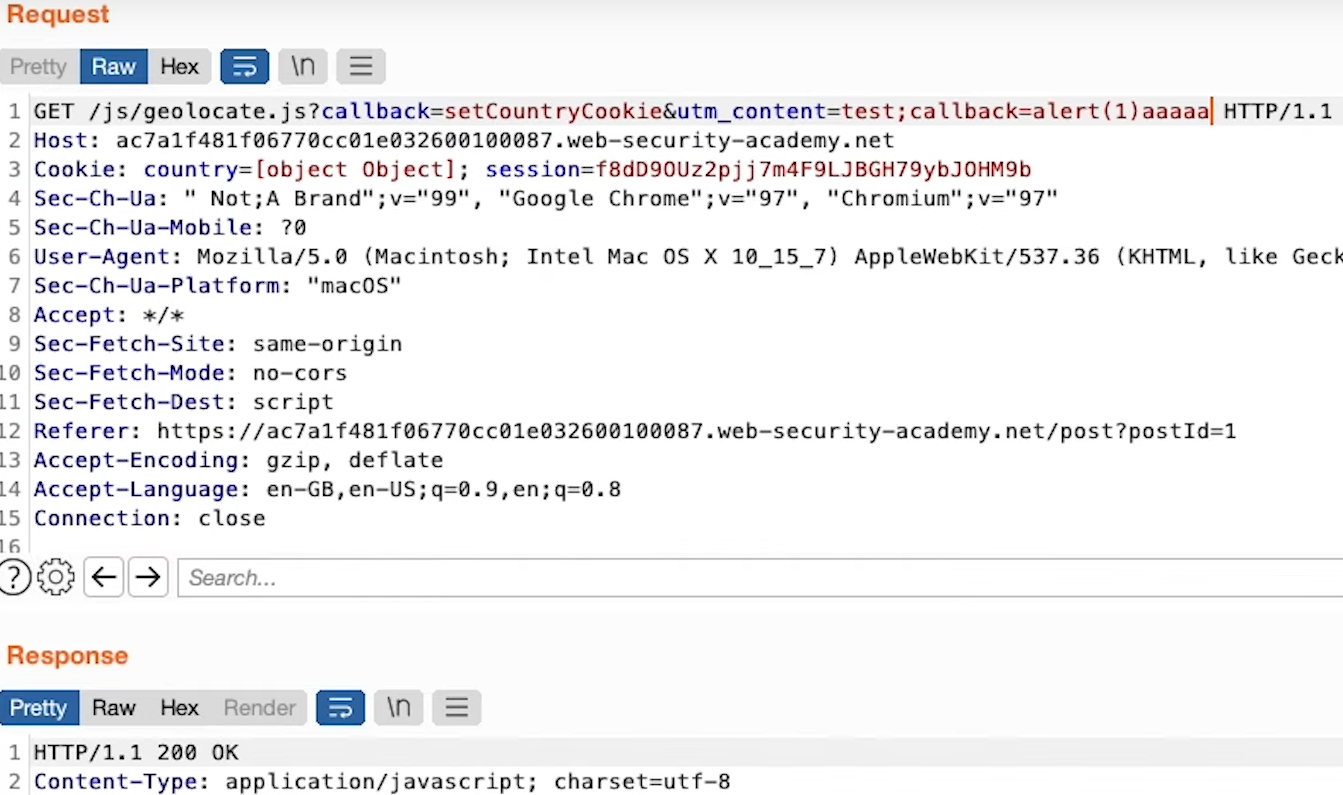
The application is vulnerable to web cache poisoning because it excludes a certain parameter from the cache key. There is also inconsistent parameter parsing between the cache and the back-end. We will try to poison the cache with a response that executes alert(1) in the visitor's browser.

**PROCEDURE**

1. Open the web application and in the BurpSuite’s Proxy tab send the GET request for **/js/geolocate.js?callback=setCountryCookie** to BurpSuite’s Repeater and study it.
2. Using the **Param Miner’s Rails parameter cloaking scan** we see that the **utm\_content** parameter is supported and is also excluded from the cache key.
3. Notice that if we use a semicolon (;) to append another parameter to utm\_content, the cache treats this as a single parameter. This means that the extra parameter is also excluded from the cache key.
4. Observe that every page imports the script **/js/geolocate.js**, executing the callback function **setCountryCookie()**.
5. Notice that we can control the name of the function that is called on the returned data by editing the URL’s callback parameter. However, we can't poison the cache for other users in this way because the parameter is keyed.
6. Study the cache behavior. Observe that if we add duplicate callback parameters, only the final one is reflected in the response, but both are still keyed. However, if we append the second callback parameter to the **utm\_content** parameter using a semicolon, it is excluded from the cache key and still overwrites the callback function in the response.
7. According to the way our injected query parameter we will craft an exploit string which will break out of that tag and trigger our alert by executing arbitrary JavaScript.
8. Append the crafted exploit as shown in the Payload in the URL.
9. Send the malicious request after removing the cache buster parameter and keep replaying the request until we see our exploit server URL being reflected in the response and **X-Cache: hit** in the headers.

**PAYLOAD**

/js/geolocate.js?callback=setCountryCookie&utm\_content=poison;callback=alert(1)

**PROOF OF CONCEPT**

**REMEDIATION**

1. **Consistent Parsing:** Ensure that both the cache and the backend parse parameters consistently. Avoid situations where one system can interpret input differently than another. This means if the backend sees two parameters, the cache should also see two parameters.
2. **Strict Input Validation:** Implement strict validation rules for the input, especially for parameters that are reflected in JavaScript code. Reject any suspicious or malformed input.
3. **Avoid Reflecting Input:** If possible, avoid reflecting user input in the response, especially in locations where it can be executed as code.
4. **Comprehensive Cache Key Generation:** The cache key should include every aspect of the request that can influence the response. In this scenario, the utm\_content parameter should not be excluded from the cache key.
5. **Avoid Directly Embedding User Input:** Especially in scripts or other executable contexts. If user input is needed in a script, consider passing the data in a way that it can't be executed, like through a data attribute in HTML which the script then reads.
6. **Sanitize Input:** Ensure that any input, especially those that can be reflected, is sanitized to prevent potential script injections. Escape characters that have special meanings in JavaScript or HTML.
7. **Use Safe Defaults:** When caching, the default behavior should be safe. If a parameter isn't recognized, the safe option is not to cache the result rather than omitting the unrecognized parameter from the cache key.
8. **Secure Cache Configuration:** Periodically review and update the cache configuration to ensure that it remains secure against evolving threats.